Singh, Gagandeep

Professor Phil Tayco

Data Structures

April 06, 2019

Analysis of the Doubly- Ended-Doubly-Linked-List

1. **void insertFront(int n)**

**Worst Case:** When the list is not empty.

**Big-O-Notation:** As the size of the list grows, the function still works on *O(1)* i.e. (O)constant. This is because the number of the significant operations of comparisons remains the same, which is one!

**Comparison of DEDLL with Standard LL**

Both DEDLL and Standard LL stay at O(1).

1. **void append (int n)**

**Worst Case:** When the list is not empty.

**Big-O-Notation:** As the size of the list grows, the function still works on *O(1)* i.e. (O)constant. This is because the number of the significant operations of comparisons remains the same, which is one as we have a pointer pointing to tail!

**Comparison of DEDLL with Standard LL**

For standard LL, the function works on O(n) as it would require traversing to reach at the end. However, with DEDLL, the function works on O(1) which is due to the tail pointer pointing to the end of the list i.e. tail.

1. **Node search (int n)**

**Worst Case:** When the list size is greater than 2 & item n is not in the list.

**Big-O-Notation:** As the size of the list grows, the function works on specifically *O(n)+O(1)*, which comes down to categorically *O(n)*. This is because the traversing node would have to traverse n (which is the size of the list) number of times. The important point here to note is that O(1) comes here from the last comparison that the list would have to go no matter how long the list grows.

**Comparison of DEDLL with Standard LL**

Categorically both DEDLL and Standard LL works on O(n), however, DEDLL has one comparison extra which makes it O(n)+O(1) specifically.

1. **boolean update (int o, int n)**

**Worst Case:** When the list size is greater than 2 & item n is not in the list.

**Big-O-Notation:** As the size of the list grows, the function works on specifically *O(n)+O(1),* which comes down to categorically *O(n).* This is because we use search function to find the item n in the list, and we know that search function works on *O(n)+O(1).*

**Comparison of DEDLL with Standard LL**

Categorically both DEDLL and Standard LL works on O(n), however, DEDLL has one comparison extra in search which makes it O(n)+O(1) specifically for the update.

1. **boolean delete (int n)**

**Worst Case:** When the list size is greater than 2 & item n is not in the list.

**Big-O-Notation:** As the size of the list grows, the function works on specifically *O(n)+O(1)*, which comes down to categorically *O(n)*. This is because the traversing node would have to traverse n times (which is the size of the list) number of times. The important point here to note is that O(1) comes here from the last comparison that the list would have to go no matter how long the list grows.

**Comparison of DEDLL with Standard LL**

Categorically both DEDLL and Standard LL works on O(n), however, DEDLL has one comparison extra which makes it O(n)+O(1) specifically.

1. **void printMiddle ()**

**Worst Case:** When the list size is greater than 2.

**Big-O-Notation:** As the size of the list grows**,** the function works specifically on *O(n/2)* which comes down to categorically *O(n)*. This is because to find the middle of the list, the traversing node would have to traverse n/2 times (which is half the size of the list).

**Comparison of DEDLL with Standard LL**

Both DEDLL and Standard Linked List works on categorically O(n).

1. **void removeDuplicates ()**

**Worst Case:** When the list size is greater than 2 and there are no duplicates i.e. the size of the list remains the same while the traversing is in process.

**Big-O-Notation:** As the size of the list grows, the function works on specifically *O(n)\*O(n-1)* which comes down to categorically *O(n^2).*This is because as the node traversing from head proceeds, the node traversing from tail to check out for the duplicates traverses (n-1) times.

**Comparison of DEDLL with Standard LL**

Categorically, both DEDLL and standard linked list works on O(n^2). However, due to the use of traversing node from the tail (in my algorithm), DEDLL works specifically on O(n)\*O(n-1).

1. **void deleteAll()**

**Worst Case:** List size is greater than 1.

**Big-O-Notation:** As the size of the list grows, the function works on specifically and categorically *O(n)***.** An important point to note here is that even when traversing through the list, delete(int n) function is used, which is of the order O(n), the overall function still remains on O(n) as a special case of the delete(int n) (which is when the item is in the first node) is used which is basically O(1).

**Comparison of DEDLL with Standard LL**

Both DEDLL and Standard Linked List works on O(n) categorically.

1. **void reverse ()**

**Worst Case:** When the size of the list is greater than 2.

**Big-O-Notation:** As the size of the list grows**,** the function works on categorically *O(n).*

**Comparison of DEDLL with Standard LL**

Both DEDLL and Standard Linked List works on O(n) categorically.

**Conclusion**

Since, both DEDLL and standard LL have same Big-O-Notation for almost every function except the append.   
A data structure or a situation that would require to add the items in the list from both the ends would be benefited from DEDLL. In addition to this, the search, update, and delete functions would show an improvement in the average cases for DEDLL.

However, the programming code for DEDLL becomes little more complex to implement than that of the standard LL.